FLORIDA SOUTHERN COLLEGE

CSC 4410: Operating Systems
(Spring 2023)

Syllabus

LMS

Teachers
Kyle

Assignments
Project 0

Other Pages
Kyle's Teaching
Kyle's Schedule
Kyle's Resources

Project 0: Semaphores

Assigned: Fri Jan 20 2023
Due: 10:00:00 AM on Fri Feb 17 2023
Team Size: 1-3
Language: Java
Out of: 100 points

In this project, you will implement a Thread-safe semaphore that can handle multiple tokens. You will reuse this code in future projects, so you want to make it as optimal as possible. Remember: you are not allowed to use the built-in Semaphore class in your implementation.

Part 0, 0 points: Create a new Java class, `MySemaphore.java`. Add a commented header to the top of your file that includes the names of everyone in your group. E.g.:

```
/**
 * Implements a multi-token semaphore.
 *
 * Authors: XXXXXXXXX
 */
```

Part 1, 0 points: Add the class header to your file:

```
public class MySemaphore {
    ... the code in the class will go here...
}
```

Part 2, 0 points: Add an integer field to your semaphore. I named mine numTokens:

```
    //the number of tokens available
    private int numTokens;
```

Part 3, 0 points: Add a constructor that takes an integer representing the number of tokens available when it's created. Users of your code should be able to create a semaphore like this:

```
MySemaphore s = new MySemaphore(2);
```

Part 4, 0 points: I'm not going to grade it, but I highly recommend you make a toString method, as you should when making a new Java class.

Part 5, 0 points: You will also need a getNumTokens() method that returns the number of available tokens in the semaphore.

Part 6, 0 points: Now it's time to write the two main methods: p() and v(). I'll give you a bad version of p() which you can update later. Note: p is short for the Dutch word *proberen*. This method blocks until the number of tokens the semaphore has is or becomes positive. When it's done blocking, it removes a token and then is done. Here's a very bad way to implement this that unnecessarily waits and is not thread safe:

```
public void p() {
    while(this.numTokens <= 0) {
        try {
            Thread.sleep(2000);
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
    this.numTokens --; //removes one token
}
```

Part 7, 0 points: Add a new method, `v()`, which is short for the Dutch word *verhogen*. This method represents the return of a token to the semaphore, so it should increment the field. It is okay for the number of tokens to go above the original number of tokens. That sounds simple, but it still needs to be thread-safe. Here is a terrible version of it that you can start with:

```
public void v() {
    this.numTokens++;
}
```

Part 8, 0 points: You should now be able to test your code with the `SemaphoreTester.java` code I wrote. I will use this code to test your submission, but it isn't enough to check that your code is actually thread-safe! I have set up assert statements in there, so if you want to make sure you're passing the tests, run it with the -ea flag, like this:

```
java -ea SemaphoreTester
```

This will cause it throw exceptions if it ever fails any of the checks I've written.

Part 9, 0 points: Here's one possible sequence of steps that could cause a race condition. There is a semaphore with 1 token and two threads are simultaneously calling the `p` method.

1. The first thread reaches the condition of the while and evaluates it to be false.
2. The second thread reaches the condition of the while and also evaluates it to be false.
3. The second thread performs the subtraction, changing the number of tokens to 0.
4. The first thread performs the subtraction, changing the number of tokens to -1, an illegal value for the Semaphore.

Part 10, 10 points: You need to modify your code so that it is thread-safe; i.e. no race conditions can occur. In order to do this, you can use two Java synchronization tools:

- The synchronized keyword, which enforces that only one thread can be executing that method at a time.
- The wait(), notify(), and notifyAll() methods that are built in to every Java object. Reminder: you can only execute these inside of a synchronized method!

You cannot use another Semaphore class, whether it's built-in or one you find. Nor should you look up code or pseudocode or any other explanation for how a Semaphore works! If you're using synchronization tools in a meaningful way (even if it's not perfect) you'll get the points for this part.

Part 11, 10 points: In addition to making sure your code is correct, you also want to improve performance (speed). Make sure that there are no calls to `Thread.sleep`, nor `System.out.println` in your Semaphore class.

Part 12, 20 points: Ensure that your semaphore doesn't block on calls to `p` when there are tokens available. This part is tested in my SemaphoreTester code. If you pass those tests, you'll get credit for this.

Part 13, 20 points: Make sure that your semaphore blocks in `p` when there are no free tokens. This part is tested in the testing file. If you pass those tests, you'll get credit for this.

Part 14, 20 points: There is no efficient test that will always be able to find all the race conditions. (Detecting deadlocks is PSPACE-hard.) Nevertheless, I wrote a stress test to try to kill your Semaphore in my testing code. If your code passes those final stress tests (with whatever parameters I choose) then you will earn the points for this.

Part 15, 20 points: No possible race conditions or deadlocks! I am going to look at your code and see if I can find a potential race condition or deadlock. If I find one or if your code doesn't pass the previous part, then you won't earn points for this part. I am willing to look at your code beforehand and see if I can find problems, but:

- Your code should be well-written for readability. Follow regular conventions (e.g. proper indentation and good variable names).
- It's generally better to ask me in person so I can explain it by pointing and gesturing at the code (and writing on the whiteboard), and
- I expect that shortly before the project is due, many people will come and start asking me to check their code. Doing this check can take a non-trivial amount of time. Start early on this! Teams usually don't get these points because they didn't leave enough time for sufficient rounds of trial-and-error with feedback from me. It is on you to come early and often enough to get this working, not the other way around!

Part 16, 0 points: If you aren't hitting the time goals you want, I recommend bringing the code to me to ask for suggestions to get it to run faster. If you're not sure where you can minimize bottlenecks and have safer multi-threading, I can help!

Submitting your Project:

Choose a team name that no one else will choose. Put all the source code files into a folder named *teamName*Project0 then zip the folder into a .zip or a .tar.gz file (*no other file formats will be accepted*). Finally, upload the zipped folder to the Canvas

assignment. If you fail to follow these directions, I will ask you to redo it and you will lose points as though you hadn't turned it in until I get the resubmission. I'll be looking for the following files in your folder:

- `MySemaphore.java`